Andreas Wegelius
15168
SAE INSTITUTE, LONDON

GAMES PROGRAMMING

DATA DRIVEN DUNGEON GENERATION

The design and implementation of dungeon PCG

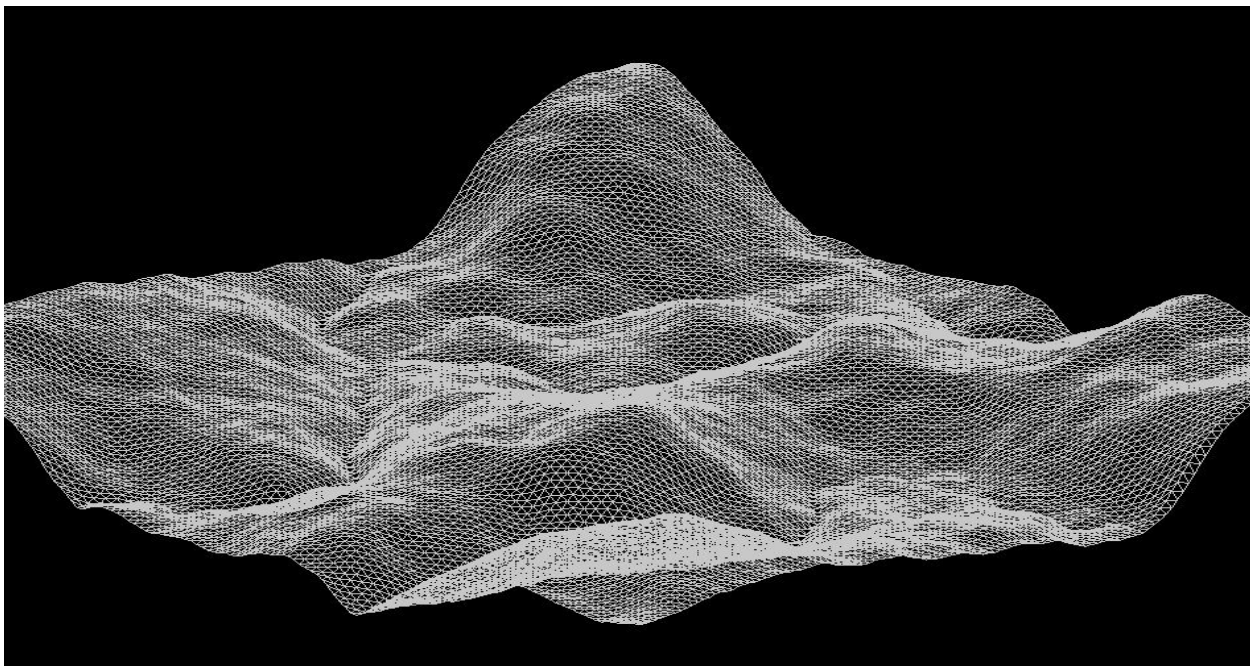


Table of Contents

Abstract

This report outlines the main aspects of the methodology, design and implementation used for the creation of a modular dungeon generator that combines procedural content generation, random content generation and user driven content generation. The final outcome is a reliable compilation of scripts that work seamlessly and produce a customizable, extensive and visually appealing dungeon.

Methodology

Procedural Content Generation

Procedural Content Generation (PCG) is the algorithmical process of creating game content with limited or indirect user input. It is a process that can be applied to many different elements within a video game. In January 2016 , Richard Moss (2016) from the popular video game development website “Gamasutra” explored a few uses of PCG in popular video games:

- Paradox Interactive’s *Crusader Kings II* uses PCG for family trees that emerge dynamically from generative personalities.
- Monolith Productions’ *Shadow of Mordor* uses PCG to generate more interesting enemies, with multiple traits and personalities.
- Sid Meier’s *Civilization* series uses PCG to create realistic earth-like environments with islands, lakes, seas and mountains.

The Modular Dungeon Generator will be a PCG-based system that will create dungeons for players to navigate within. (Shaker, Togelius and Nelson, 2016, p. 1-4.)

Algorithms

According to the Merriam-Webster Dictionary (2015), an algorithm is a set of steps that are followed in order to solve a mathematical problem or to complete a computer process. Algorithms are essential in Procedural Content Generation, they allow the computer to independently process and calculate an output that can then be interpreted and represented further. Various algorithms will be coded within the Modular Dungeon

Generator to allow it to create a complex and enjoyable environment for the player to interact within.

The various algorithms used will work side by side and in a specific order, they will run a sequence of checks and validations in order to populate a data structure (`dictionary`) containing positions and types. The order of execution of the different algorithms will determine the layout of the final dungeon and the order of execution can be manipulated by the user, within it's limitations.

In order to create a dungeon environment for the player to venture within, we must declare certains limits in order to enclose the dungeon; making it finite and preventing the player from venturing beyond it's bounds. This consists of creating an entrance and an exit, a *main* path, *random* paths that divert from the *main* path and then potential rooms within the dungeon itself. All of these require different algorithms that work hand-in-hand in order to be create a realistic final product.

The dungeon generator will be able to create *random* corridors and paths but will also be capable of generating rooms randomly or with hard-coded code snippets, these rooms will then be connected to the *main* path.

The various algorithms will be working together to populate a 2D array/grid. Once the array has received all the input required, it will be passed onto the following script which will instantiate the models.

Data Structures

Data Structures are compositions of data types which form some form of systematic organization (Sacheva, 2011, p. 5).

These facilitate the control of organized data and using data structures within languages like C++ or C# allows the programmer to create more complex scripts or algorithms whilst remaining efficient and methodical.

For instance, in the Modular Dungeon Generator, the creation of various Data Structures is essential in order to hold key variables and characteristics within active objects to be easily and properly interpreted by the instantiation script that receives key data.

However, various useful data structures are already provided within the libraries we have included in the project, the following derived data structures will be used side by side with our algorithms:

- **List** - *The List a data structure that holds items in a specific order, a list is dynamically sized and will shrink and expand depending on how many elements are contained within it.* Lists will be used to contain data and information that will be reviewed before shifting the data to the main Array.
- **Array (2d)** - *The 2D Array is a data structure that holds items in a grid like form, where items can be accessed using column and row identifiers.* The 2D array will hold all of the instances of the created data structures containing module information.
- ***Queue*** - *The Queue is a data structure where the data enters at the rear of the list and is removed from the front of the list, they store items in the order in which they occur.* Queues will be used in order to efficiently create the *random* paths that derive from the *main* paths. (McMillan, 2005, p. 80.)
- **Dictionary** - The dictionary is a data structure that associates a key with a value, it will be used to store all the tile information needed for the dungeon generation, using a position as key and a *type* as value.

Data Storage

Data Storage is an important factor to keeping an efficient, optimal and fast execution of the code required to generate the dungeon.

The data structures created for this project will hold various characteristics of each grid element that would be placed. A parent data structure will hold all of our grid elements; this specific data structure must be able to store thousands of objects and have a fast look-up speed.

The three data structures that could be adequate for the project are as follows:

- 2D Array - Arrays are solid structures that are able to contain as many elements as needed and have a fast look-up time due to it's fixed size. However, arrays are not dynamically expandable, therefore the process must "reserve" a large amount of space from memory and can not be modified unless a new array is created and the information from the previous is transferred over. This is not ideal due to the processing power and time required.
- Linked List - Linked lists are another alternative for data storage, the need to reserve memory (as for an array) is removed due to the nature of a linked list (each entry holds reference to previous and subsequent entries). However, this method is not ideal due to it's linked nature, it will allow a quick look-up for quick processing, but will not provide fast enough look-up for editing, manipulate or comparing multiple grid elements.
- Dictionary - The Dictionary is an associative array data structure that holds data in key-value pairs, it is easy to add data to, is dynamically expandable, and has a fast look-up speed whilst searching for an object either by key or by value.

The fastest and most efficient data structure is the Dictionary and it will be used for the Modular Dungeon Generator.

The Tile information will be stored in it's own data structure (TileType) that will be stored within the dictionary, this data will then be interpreted by the instantiation script.

The Pixel information read by the script are best stored within a simple 2D array, this method is preferred due to the game engine's library functions that allow textures to be read easily and have their colour information stored easily within 2D arrays (Unity Technologies, 2016).

The Asset objects which will be placed by the instantiation script are best stored within simple Lists. These lists are editable by the user and each asset is easily interchangeable via GUI for maximum customization.

Scripts for PCG and Instantiation

In order to have an efficient, reliable and speedy pipeline for the dungeon creation, the best approach would be to keep the Procedural Content Generation script separate from

the Instantiation script. Having separate scripts will guarantee a more controlled programming process for bug-testing, optimization and will differentiate the customizable elements of the software from the algorithm, and other static, non-manipulative elements.

Design

General Approach

The design approach taken in order to create the Modular Dungeon Generator consists of multiple steps. The software will first gather all the user provided data by interpreting the user fed texture file and all user declared variables and preferences.

The user will be able to provide customization preferences to the generator via both editable variables and a texture file :

- The dungeon size - The overall size of the dungeon, can also be assigned a random range.
- Entrance (from texture) - A starting position determined by the user via the texture input.
- Places of Interest (from texture) - The positions of desired chambers determined by the user via the texture input.
- Chamber size - This allows the manipulation of chamber sizes within the dungeon, can be assigned a random range.
- Wild factor - The manipulation of how direct the path will go from start to finish.
- Dungeon scale - The overall dungeon size, modifiable value used to match player size.
- Chamber corridor size - The length of the corridors leading from the main path to the chambers.

- Potential expansion from user data chambers - The possibility to have random paths derive from pre-determined chambers.
- Random chamber chance - The modifiable value that will determine the chance for random chambers to be placed.
- Data chamber chance - The modifiable value that will determine the chance for random preset chambers to be placed.
- Seed - A value that can be generated, pre-determined or stored in order to replicate the same dungeon layout,

This data will then be passed into the general generation part of the script, the preferences are applied, the main paths calculated, the random paths that divert are calculated and chambers are either introduced randomly or following the user's texture data input, depending on preference. The entrance and exit are placed according to the user provided data or the algorithm.

All the tile information will be cycled through and depending on their position, their surrounding tiles and other attributes, specific tile types will be assigned.

This new data will then be stored within the "Dictionary" data structure and passed forward to the instantiation script which will interpret all the data and instantiate all adequate modules with their respective positions, rotations, textures, models, sizes and so forth.

Finally, the player object, if plugged into the software, will be placed at the initial starting position of the dungeon.

Classes created and their characteristics

In order to keep things efficient, fast and reliable, a specific tile data class was created.

The 'PathTile' class holds various characteristics :

- Type - The type holds information representing the nature of the tile, the generation scripts first assigns types such as *"main"*, *"random"* or *"empty"* to differentiate them from each other. Once the generation is completed, a second

code snippet cycles through the tiles, checks tile-specific data and assigns more specific tile types such as *“wall”*, *“pillar”*, *“diagonal”*, *“protrude”* or *“ceiling”*.

- Position - The position holds information on where the module should be placed to successfully complete the dungeon, represented in 3 dimensional vector structures (X,Y,Z).
- Adjacent Path Tiles - A list is provided to hold data on the surrounding tiles, this is essential for an efficient and successful dungeon generation and also contributes to the customisation and realism factors of the final product. The list holds tile information in the following order : *tile above, tile to the right, tile below, tile to the left.*

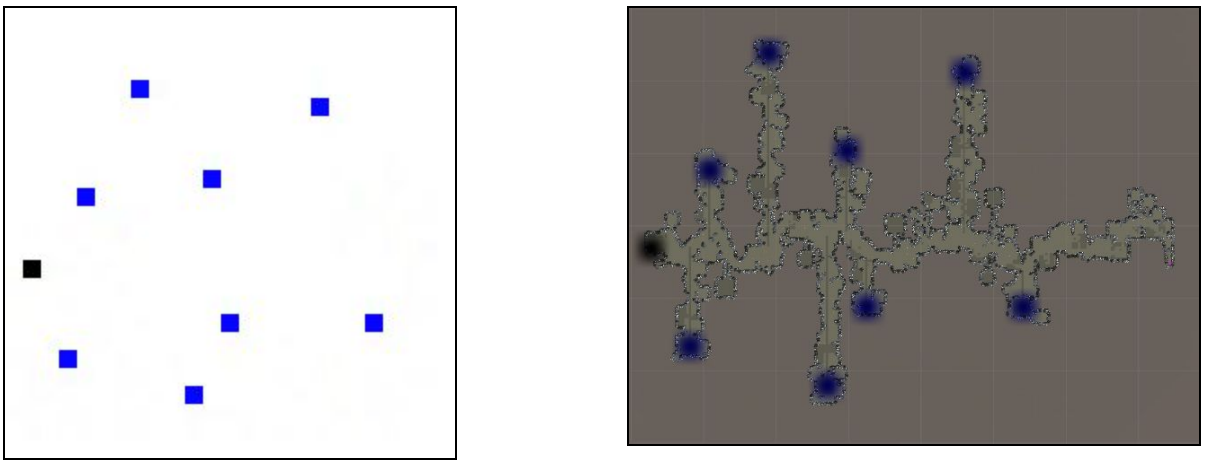
Data Gathering Script

Texture Input

The Data Gathering Script receives a texture file and proceeds to store all of the texture’s pixel information inside an *array.* The array is then scanned for colours matching blue or black.

- Black pixels (RGB : 0,0,0)- The black pixel represents the starting position of both the player and the dungeon generation. Only one black pixel is stored as a starting position, therefore the last pixel found is what will be passed forward to the generation script.
- Blue pixels (RGB : 0,0,1) - The blue pixels represent the positions of “points of interest”, these are preset hard-coded chambers. All of the positions of the blue pixels on the texture are stored in a `List`, and passed forward to the generation script.

This script can easily be expanded to detect other colours or combinations of colours and provide the dungeon with more user driven data.



Example of the pixel information contributing to the dungeon layout.

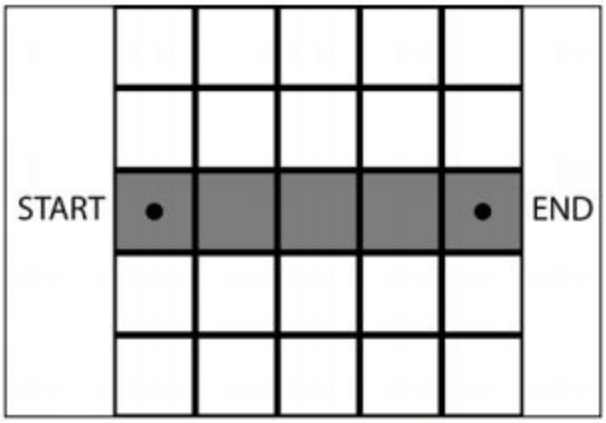
Data Script

Algorithms and order of execution

Various algorithms are executed in order to create the dungeon : the *main* path algorithm, the *random* path algorithm, the *chamber* algorithm, the *hard-coded chamber* algorithm and the *connection* algorithm. All of these contribute to the complexity and realism of the final product. However an order of execution is essential as certain algorithms rely on the data generated from others.

Main Paths

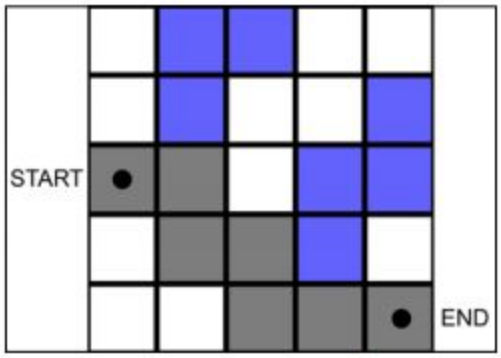
In line with the diagram below by Watkins (2016, p. 65), the main path is a simple path that connects the entrance of the dungeon to the exit. The bare-bone algorithm creates the straightest line from A to B, however using customizable variables the algorithm is able to manipulate the line to make it more wild and less direct. The wildness is achieved using PRNs (pseudo random numbers).



A diagram of the simplest essential path

Random Paths

Random paths are paths that begin and divert away from the tiles used for the main path as demonstrated in the second diagram below by Watkins (2016, p. 66). These paths move away and aim to create corridors that may end up in procedurally generated chambers or hard-coded chambers. The random paths are created using PRNs and the "queue" data structure.



A diagram of random paths in blue

Chambers

Chambers are "rooms" that are created and placed randomly. They are open areas often found at the end of corridors leading to them. Chambers are either square or circular.

Chambers are created procedurally by an algorithm that expands an area by changing the TileType of surrounding tiles, the chamber size is a customizable variable.

The corridors are generated using the “Connection Algorithm”.

Hard-coded chambers

Hard-coded chambers are square “rooms” placed either by user data via the texture file or randomly. Their size can be predetermined or within a random range. They usually have corridors leading to them, or are attached to the main path.

Hard-coded chambers are small snippets of code that when executed will create pre-determined chambers with specific tiles in specific positions, creating a less procedurally-generated outcome, ultimately contributing to the realism of the final product.

The corridors are generated using the “Connection Algorithm”.

Connection Algorithm

The connection algorithm is simple, it concludes the position of the chamber relative to the main path and runs a path down until connected. The results are straightforward and blocky but when combined with the random path algorithm, it can deliver a complex corridor.

Instantiation Script

The instantiation script simply receives all the grid information from the populated dictionary. The data, which pairs position with tile types is interpreted and the adequate assets are placed accordingly.

Various Modules

The instantiation script relies on modules to function correctly, these modules are art assets that will be placed according to the data interpreted from the generation script:

- Walls - These are the tiles that make up the walls within the dungeon.
- Pillars - These are the tiles that make up the walls that are not next to other walls.
- Diagonal - These are the tiles that make up the diagonal walls, to remove the “blockiness” of the dungeon

- Protrudes - These are the tiles that extend from walls and are only connected to one wall.
- Ceiling - These are the tiles that close off the top of the dungeon.
- Addons - These are decorative items that can be placed randomly on specific tiles.

Module-specific attributes

The various variables associated with each tile will allow the instantiation script to position, rotate, scale and stylize each asset without having to have different assets for each. This contributes to optimizing performance, speed and efficiency. The attributes are as following:

- Rotation
- Location
- Size
- Style

Player slot and positioning

By providing an input for the Player, the user can have their player automatically positioned at the script's concluded starting location.

Using the generator

The generation process takes in consideration all of the predetermined variables determined by the user, however if some variables are inaccessible or undetermined, the generator will randomize the values.

Setting up the generator

The generator's package file (.unitypackage) is easily imported into the Unity game engine from the "Asset" menu.

Once imported, the *Generator* prefab is introduced into the scene either by dragging and dropping the prefab into the Scene window or into the Hierarchy.

By clicking on the *Generator,* the customizable variables are made available in the *Inspector.* The generation is executed by clicking the Play button.

In order to manipulate the variables, the engine must **not** be in Play Mode.

Manipulating variables

General Dungeon Variables

- Max Bound - This is the size of the entire dungeon, it can also be set to randomly pick a size from a random range.
- Start Position - The start position can either be set using the user's data texture file or randomly picked by the engine.

Dungeon Size and Scale Manipulators

- Random Range Activator - A boolean toggle is used to switch between a predetermined or a randomized dungeon size from a range.
- Dungeon Scale Multiplier - A multiplier used to scale the entire up or down in order to match other assets, such as the player asset.

Preset Chamber Manipulators

- Preset Chambers Activator - A boolean toggle used to switch between having or not having hard-coded chambers instantiated.
- Preset Chamber Size - Predetermined size of chamber, will be overrun if a random range is selected.
- Preset Chamber Random Range Activator - A boolean toggle is used to switch between a predetermined or a randomized hard-coded chamber size from a range.
- Corridor Size - The predetermined size of the corridors leading to randomly generated hard-coded chambers.
- Data Chambers Expansion Activator - Manipulates the order of execution of the random paths script, allowing the dungeon to create random paths from user driven hard-coded chambers.

Customizable Generation Settings

- Wild Factor - Variable manipulating how direct the path will go from start to finish.

- Random Chamber Chance - Variable manipulating the chance to randomly create a chamber at the end of a random path.
- Preset Chamber Chance - Variable manipulating the chance to randomly create a hard-coded chamber at the end of a random path.
- Chamber Size - Variable representing the size of randomly generated chambers.

User Texture Data input

- Data Texture - Input for the user created texture that holds pixel information representing starting points and points of interest.

Dungeon Seeding

- Seed - A variable that the user can determine or store in order to recreate the same dungeon. Setting this to 0 will allow the generator to generate a random seed.

Texture Input

The texture input is easily plugged in by simply dragging and dropping it into the *Data Texture* field within the *Generator* prefab. However the texture must be imported correctly in order to be read by the generator.

Creating the texture

Small images with less pixels are more ideal for a faster execution of the generation script. Square images are recommended in order to have the positioning of elements correctly imported.

‣ Placing a black pixel will represent the starting point, it is recommended that the black pixel is towards the left of the image, as the nature of the pathfinding algorithms go from left to right.

‣ Placing a blue pixel will represent a Point of Interest (hard-coded chamber).

Importing the texture

A texture file is easily dropped into the project file for Unity to acknowledge it. However the import settings must be edited for it to be read properly.

By clicking on the texture file, the *Import Settings* are displayed in the *Inspector:*

- The *Texture Type* must be set to *Advanced*.
- The *Non Power of 2* must be set to *None*.
- The *Read/Write Enabled* box must be ticked.

Implementation

Populating the grid

In order to create the dungeon and place the modules adequately, the class `PathTile` is used by the algorithms to deduce which tiles should be where, this class will take various `TileTypes` such as main, random and empty depending on its nature. It also holds a function called `getAdjacentPath` that will check the surrounding `PathTiles` and store their position in a `List` in order to easily check their types.

The position of each `PathTile` is stored as a Vector2 with an X and Y value, and the `TileType` varies depending on the nature of the tile. The various `TileTypes` are as follows and are set up as enumerators:

```
public enum TileType { essential = 1, random = 2, empty = 3, wall = 4, pillar = 5,
diagonal = 6, diagonal2 = 7, diagonal3 = 8, diagonal4 = 9, protrude1 = 10, protrude2
= 11, protrude3 = 12, protrude4 = 13, chamber = 14 };
```

The `getAdjacentPath` is a simple function that has multiple `if` functions that will store the surrounding tiles into a `List`.

```
public List<Vector2> getAdjacentPath(int minBound, int maxBound, Dictionary<Vector2,
TileType> currentTiles) {
            List<Vector2> pathTiles = new List<Vector2> ();
            //TOP
            if (position.y + 1 < maxBound && !currentTiles.ContainsKey(new
Vector2(position.x, position.y + 1))) {
                pathTiles.Add(new Vector2(position.x, position.y + 1));
            }
...
```

The Vector2 tile positions and the TileType of each tile *(essential, random, empty)* is deduced by various algorithms and introduced into the dictionary, the dictionary is then cycled through and the more specific final TileTypes are added *(wall, pillar, diagonal, protrude, chamber..)*.

User Texture Data interpretation

The user is able to manipulate certain variables and positions of specific aspects of the dungeon. This is achieved by introducing a texture file that holds various colours.

The script will detect blue and black pixels, interpret their meaning, scale them to the grid and alter the variables accordingly.

The *`gatherData`* function simply deduces the width and height of the texture and stores it. The Unity built-in function *`.GetPixels()`* will store all of the pixel and their colour information into a *`Color Array`*, this colour array is then cycled through and if the colour matches what the script is searching for, that specific pixel's position on its texture is stored and passed on to the general generation script.

```
//Storing pixel information, cycling through the array and finding specific colours
Color[] pix = dataTexture.GetPixels();
for (int row = 0; row < dataTexture.width; row++){
            for (int col = 0; col < dataTexture.height; col++){
                if (dataTexture.GetPixel(row, col).r < 0.1 &&
dataTexture.GetPixel(row,                     col).g < 0.1 &&
dataTexture.GetPixel(row, col).b > 0.9){
                    Debug.Log("Found Blue" + row + col + "added to list");
                    dataChambers.Add(new Vector2(row, col));
                }
...
```

In order for the positions to match the final dungeon's grid, a simple formula is applied to the pixels positions :

`DungeonSize / TextureSize = x;`

`Pixel Position * x = new position.`

This positional data is passed forward down the pipeline to the generation script where it will be interpreted accordingly.

Essential Paths

The *Essential Path* is the main path that will connect the entrance to the exit, the algorithm is a pathfinder that will connect a start position to the end position. In order to optimize the space and to fit the dungeon size provided by the user, the algorithm will start by default from the far left part of the grid and end at the far right.

The *BuildEssentialPath* function (if not specified differently by the user data input) will pick a random position as a starting point. This point will have a random Y position value selected between 0 and the maximum bound of the dungeon size. The X position value will be the far left of the grid.

The first tile of our *Essential Path* is then created using the random values and the dungeon's start position is decided.

```
ePath = new PathTile(TileType.essential, new Vector2(0, randomY), minBound,
maxBound, gridPositions);
startPos = ePath.position;
```

The current *PathTile's* adjacent path tiles list is then accessed and one of the tiles is randomly chosen to be the next *PathTile*. This is repeated within a *while statement* till the creation and placement of all the *PathTiles* has reached the far right of the grid.

```
while (boundTracker < maxBound) {
       // Essential Path Creation
      }
```

In order to deduce how far along the grid the current tile in question is located, a simple counter is introduced *boundTracker*. This counter is incremented every time an *if* statement check has deduced the essential path algorithm has shifted to the right.

Finally, if the algorithm has reached the far right of the grid *maxBound*, it will place a final tile and nominate it as the end position *endpos*.

Random Paths

The Random Path algorithm will cycle through all of the essential path tiles created in the previous algorithm by creating a *List* that will act as a queue and hold all of the essential path tiles. The queue is created in order to easily iterate over our main path tiles.

```
 ...
//List is used has our queue, easier to add to end and remove from front
```

```
 List<PathTile> pathQueue = new List<PathTile> ();

//copying the essential path to the pathQueue list
        foreach (KeyValuePair<Vector2,TileType> tile in gridPositions)
        {
            Vector2 tilePos = new Vector2(tile.Key.x, tile.Key.y);
            pathQueue.Add(new PathTile(TileType.random, tilePos, minBound, maxBound,
gridPositions));
        }
…
```

The algorithm will check if the *PathTiles* have tiles surrounding them, and if there are none it will have a possibility to extend a random path from the tile.

The random path tile that is created is then added to the *gridPositions* dictionary that holds all of the tile information. The tile is also added to the back of the queue in order to be reprocessed and potentially be expanded further.

This algorithm is repeated until there are no tiles remaining in the queue.

```
 ...
else if (Random.Range(0, myWildFactor) == 1 || (tile.type == TileType.random &&
adjacentTileCount > 1)){
         int randomIndex = Random.Range(0, adjacentTileCount);
         Vector2 newRPathPos = tile.adjacentPathTiles[randomIndex];
         if (!gridPositions.ContainsKey(newRPathPos)){
                    gridPositions.Add(newRPathPos, TileType.empty);
                    PathTile newRPath = new PathTile(TileType.random, newRPathPos,
minBound, maxBound, gridPositions);
                    pathQueue.Add(newRPath);
…
```

The *Random Path* algorithm also has a code snippet that holds the chance to potentially create Randomly generated chamber or a *Preset Chamber*.

Random Chamber Creation

If the *Random Path* algorithm triggers a *Random Chamber Creation*, the script will run another function that will create a chamber at the end of the random path.

The algorithm is simple and will input the positions and tile types of the chamber tiles to the *gridPositions* dictionary.

The code first randomly chooses a tile surrounding the tile passed in from the random path algorithm.

```
private void BuildRandomChamber (PathTile tile)  //Function called from random path
algorithm, passing in tile
```

A simple *for loop* is then used to cycle through all the positions of the new tiles that build the chamber, these tiles are added to the dictionary. The *for loop* takes in consideration the maximum and minimum bounds of the grid and also checks if the tile exists, and if so, it does not add it to the grid.

```
  //Loop through all tiles we need to add to chamber
        for (int x = (int) chamberOrigin.x; x < chamberOrigin.x + chamberSize; x++)
{
            for (int y = (int) chamberOrigin.y; y < chamberOrigin.y + chamberSize;
y++){
                Vector2 chamberTilePos = new Vector2 (x, y);
                if (!gridPositions.ContainsKey(chamberTilePos) &&
                 chamberTilePos.x < maxBound && chamberTilePos.x > 0 &&
                 chamberTilePos.y < maxBound && chamberTilePos.y > 0){
                    //add these new tiles to the dictionary
                   gridPositions.Add (chamberTilePos, TileType.empty);
...
```

Premade Chamber Creation

The premade chamber creation algorithm is similar to the random chamber creation algorithm however it is more complex.

The function can be called from different parts of the general code. It can be called as an additional randomly generated chamber but with more customisability or it can be called based on whether the user has provided positions for premade chambers using the texture file input.

The chamber creation function is called with two properties, the tile where the chamber should placed and a boolean representing whether the function was triggered via the random path function or the user data driven function.

```
 private void BuildPresetChamber(PathTile tile, bool dataChamber) //called with tile
information and boolean defining type of chamber (random or user)
```

The chamber creation algorithm will first check whether the user has specified random sizes, and if so it will randomly chose a number from a range and apply that as it's size.

If the data chamber is randomly assigned, the positions of the 3 adjacent tiles above and below are stored and are used to check whether the chamber is connected to other paths from above or below, this is crucial for the creation of corridors that will lead to these chambers. The corridor size is a variable that can be manipulated by the user.

If the data chamber is user data driven, the function will check whether it is located above or below the essential path and create a corridor leading to the essential path using a while loop.

```
 if (chamberOrigin.y > startPos.y){
                Vector2 tempPos = new Vector2(chamberOrigin.x, chamberOrigin.y - 1);
                while (!gridPositions.ContainsKey(tempPos)){
                      gridPositions.Add(tempPos, TileType.chamber);
                      tempPos.y -= 1;
...
```

The while loop will continuously run and create a corridor until it detects an existing tile.

In order to limit the amount and chambers and to have them separated and not connected, a quick check is made before the creation of the chamber. The check consists of checking each 4 corners of the potential chamber for existing chambers, if it finds one, it will not create the chamber.

```
  //IF THE POTENTIAL LOWER LEFT CORNER EXISTS BUT IS NOT A CHAMBER OR IF THE
POTENTIAL LEFT CORNER DOES NOT EXIST, CONTINUE
if (gridPositions.ContainsKey(new Vector2(chamberOrigin.x - presetChamberSize / 2,
chamberOrigin.y - presetChamberSize / 2)) && gridPositions[new
Vector2(chamberOrigin.x - presetChamberSize / 2, chamberOrigin.y - presetChamberSize
/ 2)] != TileType.chamber || !gridPositions.ContainsKey(new Vector2(chamberOrigin.x
- presetChamberSize / 2, chamberOrigin.y - presetChamberSize / 2)))
        {
        …
        }
```

Wall Creation and customization.

The walls are placed after all the creation algorithms have run their course, the function is called by the general code and the dictionary containing all the tile information is passed through.

The function creates a List of *`PathTiles`* that it will fill up with *`PathTiles`* that hold their own specific type.

The function cycles through all of the tiles in the dictionary and checks whether they have tiles surrounding them, if they don't it will create a *`PathTile`* of type *`wall`* and insert it into the *`List`*.

```
 List<PathTile> wallTilesList = new List<PathTile>();
        foreach (KeyValuePair<Vector2, TileType> tile in myTiles){
            //LEFT
            if (!myTiles.ContainsKey(new Vector2(tile.Key.x - 1, tile.Key.y))){
                Vector2 tilePos = new Vector2(tile.Key.x - 1, tile.Key.y);
                wallTilesList.Add(new PathTile(TileType.random, tilePos, minBound,
maxBound, gridPositions));
            }
...
```

Once all the walls are found and added to the List, the list is cycled through and all it's data is added to general *gridPositions* dictionary.

The *gridPositions* dictionary is then cycled through and the tiles surrounding each wall is checked for the potential to place *diagonal walls, pillars or protrude walls* then replaces the tile with the adequate *TileType.*

```
...
foreach (PathTile tempTile in wallTilesList){
            if (tempTile.type.ToString() == "diagonal") {
            gridPositions[tempTile.position] = TileType.diagonal;
            }
...
```

When all of the tiles have been processed, the final *gridPositions* dictionary is passed on to the board manager for module placement and interpretation.

Board Manager interpretation and instantiation of 3D objects.

The board manager is a separate script that receives the final grid Positions dictionary and places modules at their respective positions.
The script contains variable arrays of *GameObjects* ranging from *floorTiles* and *wallTiles* to pillars and *objectAddons.*

```
    //Arrays containing various objects
    public GameObject[] floorTiles;
    public GameObject[] outerWallTiles;
    public GameObject[] wallTiles;
...
```

These expandable arrays contain the models that are to be placed and rendered. The user can easily fill these arrays and have them edited and replaced.

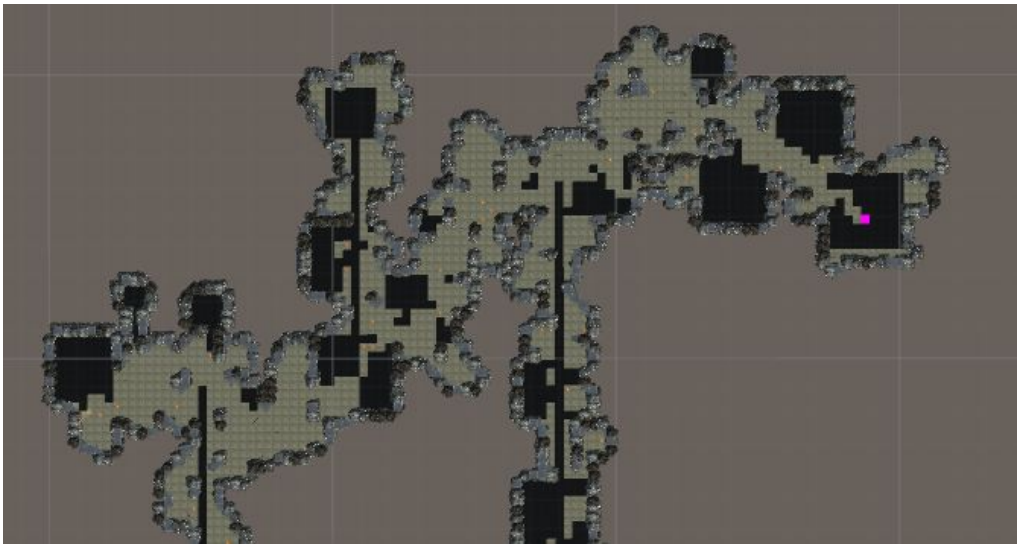
The board manager script simply cycles through the data in the `gridPositions` dictionary, checks the position and TileType. It will then randomly select an object from its corresponding GameObject array and place it at said position.

However, depending on the `TileType`, the `Board Manager` will also rotate diagonal walls to prevent the need for multiple rotation-specific walls to be created and introduced into the object arrays.

There is also a chance that floor tiles will hold `objectAddons`, which are simple, decorative models, these are placed slightly above the floor tile.

The ceiling is also placed according to every tile within the dungeon, it is a simple function that places ceiling models above the walls and floor at various heights.

Each module created in inserted within a group representing its general type, `Dungeon` or `DungeonRoof`.



The final result is a complex modular dungeon which is procedurally, randomly and user data generated.

Testing

Settings Limitations

In order for the Generator to function and run properly and with stability, restrictions were applied to control the initial variable input and set them to defaults if they are problematic.

The `PreChecks()` function will check as follows whether all variables are adequate:

- Check if the seed is determined, if not, the function will generate it's own seed.
- Check if a player prefab is plugged in, otherwise do not attempt to place a player.
- Check if the dungeon size is smaller than the general dungeon size.
- Check if the random dungeon size maximum range variable is higher than the minimum range.
- Check if the user has provided a data texture in the texture field.
- Check if the hard-coded chamber size is smaller than half of the general dungeon size.
- Check if the random hard-coded chamber maximum range variable is higher than the minimum range.

By performing such checks, and replacing illegal variables with default variables, the generation script is guaranteed to run flawlessly.

Speed of execution

The speed of execution and completion of the generation script can differ depending on the predetermined variables, whether a texture file was introduced and how many point of interest elements were imported from the texture file.

Represented below are various tests performed as well as the average time it took to complete. The average time it takes to complete for each example can differ depending on the processing power of the system running the script. Each test was performed 10 times.

Dungeon variable presets	Texture File	Average time to complete (in seconds)
Small dungeon (100x100)	none	0.995
Small dungeon (100x100) with data texture with few elements	yes	1.054
Small dungeon (100x100) with data texture with many elements	yes	1.215

Large dungeon (500x500) with data texture with few elements	yes	1.559
Large dungeon (500x500) with data texture with many elements	yes	2.471
Large dungeon (500x500) with low wild factor, low random chance and low preset chance variables	none	1.384
Large dungeon (500x500) with high wild factor, high random chance and high preset chance variables	none	1.472
Small dungeon (100x100) with low wild factor, low random chance and low preset chance variables	none	1.007
Small dungeon (100x100) with high wild factor, high random chance and high preset chance variables	none	1.022

The generator creates a dungeon layout rather quickly and no variable drastically affects the performance. However it is evident that the larger the dungeon is, the more time it takes to create and that if the texture file contains multiple points of interest, the script will take slightly longer to interpret all the data.

Bibliography

Dunn, F. and Parberry I. (2002) *3D Math Primer for Graphics and Game Development*. Plano: Worldwide Publishing, Inc.

Goldstone, W. (2009) *Unity Game Development Essentials*. Birmingham: Packt Publishing Limited.

McGrath, M. (2013) *C++ Programming*. 4th edn. Leamington Spa: Easy Steps Limited.

References

McMillan, M. (2005) *Data Structures and Algorithms Using C#.* New York: Cambridge University Press.

Merriam-Webster Dictionary (2015) *Algorithm*. Available at: http://www.merriam-webster.com/dictionary/algorithm (Accessed: 20 July 2016).

Sacheva, Y. (2011) *Beginning Data Structures Using C*. Kindle Edition. New Delhi: Yogish Sacheva. (Accessed: 18 July 2016).

Shaker, N., Togelius J. and Nelson, M. J. (2016) *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. [Online] Available at: http://pcgbook.com/ (Accessed: 20 July 2016).

Unity Technologies (2016) *Scripting API: Texture2D*. Available at: https://docs.unity3d.com/ScriptReference/Texture2D.GetPixels.html (Accessed: 20 July 2016).

Watkins, R. (2016) *Procedural Content Generation for Unity Game Development*. Birmingham: Packt Publishing Limited.

Moss, R. (2016) 7 uses of procedural generation that all developers should study. Available at: http://www.gamasutra.com/view/news/262869/7_uses_of_procedural_generation_that_all_developers_should_study.php (Accessed: 21 July 2016).